CADIFF: SCAFFOLD FIRST, DENOISE THE REST FOR CAD PROGRAM GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Natural-language-to-CadQuery generation requires a model to produce executable programs whose parameters, feature implementations, and build logic remain consistent across distant code regions. Existing autoregressive methods resolve these decisions sequentially according to token order, while generic diffusion models jointly denoise program organization and implementation in the same masked space. We identify this mismatch between token-level uncertainty resolution and the different scopes of CAD construction decisions as the *construction-level abstraction gap*. To address it, we propose **CADiff**, a construction-first diffusion framework that resolves shared program organization before concrete implementation. CADiff first predicts a compact construction layout containing interface-level structure and implementation-span capacities, renders it into an anchored program canvas, and then performs anchor-aware bidirectional denoising over the remaining code. We further introduce **CADConstruct**, a construction-aware benchmark with dimension-grounded requirements, executable CadQuery programs, and explicit construction structure. Across easy, medium, and hard subsets, CADiff consistently outperforms strong autoregressive and diffusion-based baselines in executability, feature recovery, and geometric fidelity, with its advantage increasing on more complex programs. Ablations support the complementary roles of layout conditioning and dependency-aware denoising, while Stage-I diagnostics show over 98% canvas validity and budget coverage. Zero-shot evaluation on ExeCAD further demonstrates improved cross-dataset transfer over CADConstruct-trained baselines.

1 INTRODUCTION

Computer-aided design (CAD) underpins modern engineering, product development, architecture, and manufacturing. As AI-assisted design continues to advance, translating natural-language specifications into editable parametric models has emerged as an important research problem (Khan et al., 2024; Xie & Ju, 2025; Guan et al., 2025; Niu et al., 2026). Natural-language-to-CadQuery generation addresses this goal by synthesizing executable CAD programs from textual modeling requirements. Unlike direct mesh generation, these programs encode the parametric construction logic underlying the final geometry, thereby preserving editability, inspectability, and geometric verifiability.

Despite recent progress, natural-language-to-CadQuery synthesis requires construction-level consistency across distant code regions: a feature must agree with the parameters it consumes, the intermediate solid it modifies, and the surrounding build logic. Yet existing work predominantly casts the task as left-to-right generation of a flat code sequence (Xie & Ju, 2025; Guan et al., 2025; Niu et al., 2026). This factorization resolves decisions according to textual position even though different decisions have very different scopes: some affect only a local expression, whereas others must remain consistent across multiple functions and distant code regions. Consequently, locally plausible choices can conflict with later interfaces, variable bindings, or construction operations, producing programs that are syntactically plausible but non-executable or geometrically incorrect. Existing benchmarks commonly represent target programs as relatively monolithic scripts, leaving such construction organization largely implicit (Xie & Ju, 2025; Guan et al., 2025).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

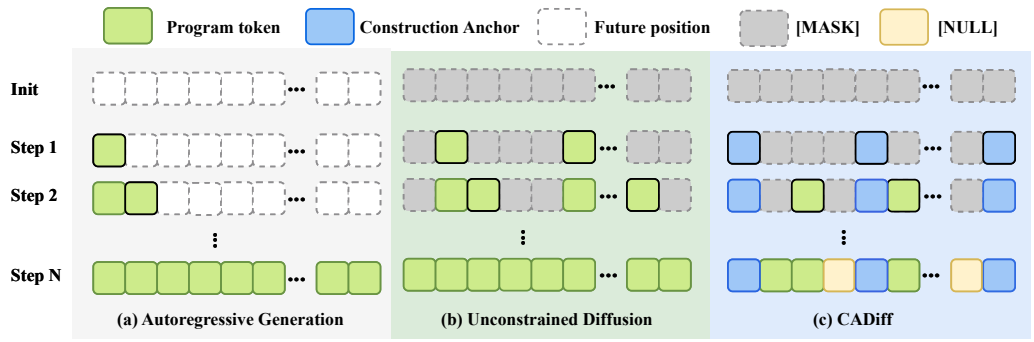

Figure 1: Comparison of uncertainty resolution in CadQuery generation. Autoregressive decoding proceeds left to right, while unconstrained diffusion jointly denoises the full program. CADiff first establishes interface-level construction anchors and span capacities, then denoises the remaining implementation tokens under these fixed conditions, with NULL tokens absorbing unused capacity.

Diffusion language models offer a natural alternative to autoregressive generation: bidirectional denoising allows distant program regions to condition one another without rigid left-to-right commitments (Nie et al., 2025; Ye et al., 2025; Xie et al., 2025). Yet generic diffusion moves toward the opposite extreme, leaving construction organization and implementation details jointly unresolved in the same masked program space. Bidirectional context allows these regions to interact, but does not by itself specify which shared decisions should be established before the implementations that depend on them. The model must therefore infer globally shared program organization and local implementations simultaneously. In both direct autoregressive and unconstrained diffusion generation, uncertainty is resolved primarily at the token level, without explicitly reflecting the unequal scope of CAD construction decisions. We refer to this mismatch between token-level uncertainty resolution and hierarchical construction dependencies as the **construction-level abstraction gap**.

To address this gap, we propose **CADiff**, a construction-first hierarchical diffusion framework guided by a simple principle: *shared construction organization should be resolved before the implementations that must conform to it*. Although compact, interface-level decisions can constrain many downstream code regions. We therefore treat such globally shared choices as *construction forks*: they do not determine the final implementation, but resolving them narrows the set of implementations that remain compatible with the intended program organization. As illustrated in Figure 1, Stage I predicts a compact construction layout containing the parameter-region structure, helper-function interfaces and return bindings, their declaration order, the final result binding, and implementation-span capacities. A deterministic renderer converts this layout into an anchored program canvas. Conditioned on this construction state, Stage II generates the remaining implementation. CAD-specific single-block and cross-block denoising views further train the model to recover local implementations and coordinate dependencies across parameter, feature, and build regions. CADiff therefore resolves uncertainty in two semantic stages: first over shared construction organization, and then over the implementation choices that must remain consistent with that organization.

To support this formulation with explicit supervision and evaluation, we further introduce **CADConstruct**, a construction-aware benchmark. By explicitly organizing parameter regions, feature functions, intermediate-solid flow, and build logic, CADConstruct makes construction organization directly available for supervision while retaining complete executable programs and reference geometry. This enables evaluation at complementary levels of code executability, coarse CAD feature recovery, and geometric fidelity.

Experiments on CADConstruct show that CADiff consistently outperforms strong autoregressive and diffusion-based baselines across executability, coarse CAD feature recovery, and geometric fidelity, while component ablations support the proposed construction-first design. Since CADiff generates programs in a standardized CadQuery organization, we additionally ask whether this canonical output structure limits transfer beyond CADConstruct. Zero-shot evaluation on ExeCAD (Niu et al., 2026) shows that CADiff retains its standardized program representation while achieving improved executability and geometric fidelity on the new benchmark. These results demonstrate the broader applicability of CADiff beyond CADConstruct.

Our main contributions are summarized as follows:

- We formulate the **construction-level abstraction gap** in natural-language-to-CadQuery generation, highlighting the mismatch between token-level uncertainty resolution and the different scopes of construction decisions across a CAD program.
- We propose **CADiff**, a construction-first diffusion framework that first predicts shared interface-level program organization, renders it into an anchored program canvas, and then performs anchor-aware bidirectional denoising over the remaining implementation spans.
- We introduce **CADConstruct**, a construction-aware benchmark with dimension-grounded requirements and executable CadQuery programs that expose explicit construction structure for supervision and evaluation.
- Extensive experiments show that CADiff consistently improves executability, coarse CAD feature recovery, and geometric fidelity over strong autoregressive and diffusion-based baselines. Component ablations support the construction-first design, while zero-shot evaluation on ExeCAD demonstrates broader applicability beyond CADConstruct.

2 Related Work

2.1 CAD Program Generation

Learning-based CAD generation has explored parametric and programmatic representations beyond raw meshes, including CAD operation sequences, sketch-and-extrude programs, relational sketch graphs, parametric sketch models, and B-rep structures (Wu et al., 2021; Xu et al., 2022; Seff et al., 2020; 2022; Xu et al., 2024; Jayaraman et al., 2023). Recent work further extends CAD generation to natural-language and multimodal settings. Text2CAD generates parametric CAD command sequences from text (Khan et al., 2024); Text-to-CadQuery maps textual descriptions to executable CadQuery scripts (Xie & Ju, 2025); Query2CAD uses LLMs with feedback-guided refinement (Badagabettu et al., 2024); and CAD-Coder/CAD-RL improve CAD code generation through chain-of-thought reasoning, geometric rewards, and reinforcement learning (Guan et al., 2025; Niu et al., 2026). More recent approaches address complementary challenges: FutureCAD combines LLM-based executable CadQuery generation with B-rep primitive grounding (Li et al., 2026); and IterCAD performs iterative program repair from dimension-annotated orthographic drawings (Wu et al., 2026). CADiff focuses instead on the generation process of a diffusion language model. Rather than learning a complete structural hierarchy or refining a generated program through external feedback, CADiff predicts a compact interface-level construction layout, deterministically renders it into persistent code anchors, and denoises the remaining implementation under these fixed conditions.

2.2 Diffusion Large Language Models

Diffusion language modeling replaces left-to-right decoding with iterative denoising over corrupted token sequences, enabling infilling and bidirectional context modeling (Austin et al., 2023; Li et al., 2022; Sahoo et al., 2024). Recent Diffusion Large Language Models (dLLMs) have substantially advanced this paradigm. LLaDA demonstrates large-scale masked diffusion pre-training and instruction tuning (Nie et al., 2025), while Dream 7B supports arbitrary-order generation, infilling, and controllable decoding (Ye et al., 2025). For code generation, Dream-Coder adapts diffusion language modeling to programming tasks and exhibits flexible decoding behaviors (Xie et al., 2025).

3 CADiff Framework

3.1 Hierarchical Formulation

Given a natural-language modeling requirement x, our goal is to synthesize an executable CadQuery program y. Under our standardized program representation, each program admits a canonical decomposition into a construction layout a and a collection of implementation spans $u = (u_1, \ldots, u_K)$:

$$y = \mathcal{R}(a, u), \tag{1}$$

where $\mathcal{R}$ is a deterministic program assembly operator.

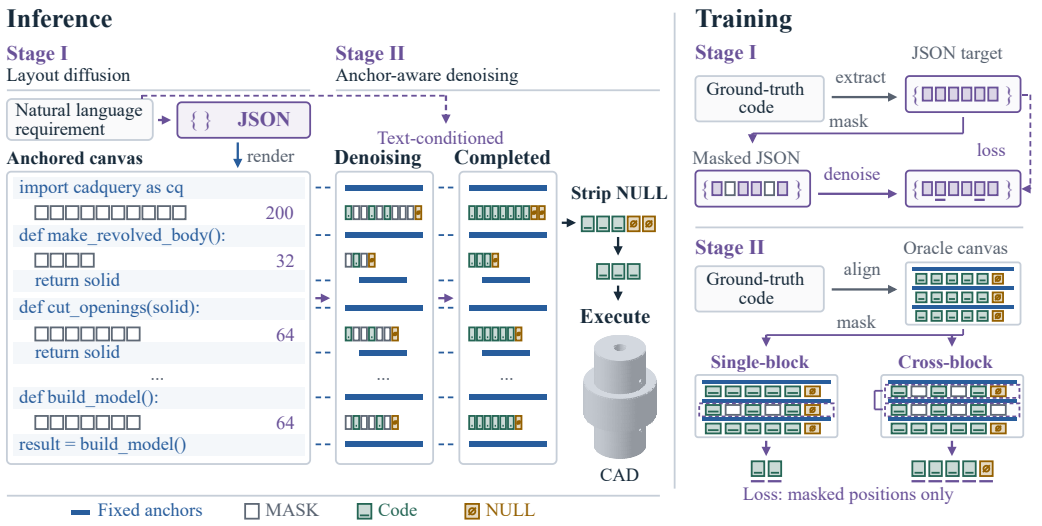

Figure 2: Overview of CADiff. **Left: inference.** Stage I predicts a construction layout, which is rendered into an anchored canvas for Stage-II implementation denoising. **Right: training.** The two stages learn layout reconstruction and anchor-aware implementation denoising, respectively.

The construction layout a is a compact structured record that specifies the parameter-region organization, helper-function interfaces and return bindings, their declaration order, the final result binding, and the token capacities of the implementation spans. Canonical import statements and the `build_model` interface are also retained as visible anchors in the serialized layout. In contrast, the implementation spans u contain the concrete parameter assignments, helper-function bodies, and the complete `build_model` body. Numerical values, derived expressions, body-internal variable flow, and the realized runtime call order therefore remain implementation decisions.

This decomposition separates uncertainty about shared program organization from uncertainty about its concrete implementation. The compact layout provides interfaces and span capacities that are shared across multiple program regions, while the implementation spans determine how individual features are realized and composed under that organization. Under this canonical decomposition, CADiff factorizes

$$p(y \mid x) \equiv p(a, u \mid x) = p_\phi(a \mid x)\, p_\psi(u \mid x, a). \tag{2}$$

For a given layout a, define

$$\Omega(a) = \{\mathcal{R}(a, u) \mid u \in \mathcal{U}(a)\}, \tag{3}$$

where $\mathcal{U}(a)$ denotes the implementation-span tuples that satisfy the interfaces and span capacities specified by a. Thus, $\Omega(a)$ denotes the set of programs compatible with the predicted layout. Stage I resolves the layout variables that establish these shared conditions, and Stage II models the remaining implementation uncertainty within the resulting layout-constrained space.

The resulting generation pipeline is

$$x \to a \to c \to \bar{y} \to y, \tag{4}$$

where c is the anchored program canvas rendered from a, $\bar{y}$ is its budget-aligned completion, and y is the final CadQuery program obtained after span postprocessing and program assembly.

3.2 Stage I: Construction-Layout Diffusion

Stage I predicts the compact program organization used to condition subsequent implementation generation, including the parameter-region organization, helper-function interfaces and return bindings, declaration order, final result binding, and implementation-span capacities. Although compact, these interface-level choices can affect multiple downstream code regions. For example, a helper signature constrains both its own body and the build routine that invokes it. We refer to such globally shared interface choices as *construction forks*.

For each training pair (x, y), a deterministic extractor $\mathcal{E}$ obtains the oracle construction layout $a^* = \mathcal{E}(y)$ from the standardized program, and a serializer $\mathcal{S}$ converts it into a compact token sequence

$$z_0 = \mathcal{S}(a^*). \tag{5}$$

Stage I follows the masked-diffusion objective of the underlying dLLM. We sample $t \sim \mathcal{U}(0, 1)$ and independently replace each token in z_0 with `[MASK]` with probability t, yielding $z_t \sim q_t(z_t \mid z_0)$. Let

$$\mathcal{M}_t = \{i \mid z_{t,i} = \texttt{[MASK]}\} \tag{6}$$

denote the corrupted positions. For brevity, let

$$p^{\mathrm{I}}_\phi := p_{\theta+\Delta^{\mathrm{I}}_\phi}$$

denote the shared dLLM backbone equipped with the Stage-I LoRA adapter. The training objective is

$$\mathcal{L}_{\mathrm{I}} = -\mathbb{E}\left[\sum_{i\in\mathcal{M}_t} w(t) \log p^{\mathrm{I}}_\phi\left(z_{0,i} \mid x, z_t\right)\right], \tag{7}$$

where the expectation is taken over training pairs, diffusion time $t \sim \mathcal{U}(0, 1)$, and the corresponding corruption process, and $w(t)$ denotes the diffusion reweighting term.

At inference, Stage I initializes a fully masked layout sequence and iteratively denoises it conditioned on the natural-language requirement x. The completed sequence is then deserialized and validated to obtain the predicted construction layout $\hat{a}$, which is passed to the deterministic canvas renderer used by Stage II.

3.3 Anchored Program Canvas

A deterministic canvas renderer $\mathcal{C}$ converts a construction layout a into an anchored program canvas

$$c = \mathcal{C}(a) = A_0\,\texttt{[MASK]}^{\,b_1} A_1 \cdots A_{K-1}\,\texttt{[MASK]}^{\,b_K} A_K, \tag{8}$$

where A_k denotes a visible anchor sequence and b_k is the token capacity of the k-th implementation span. The anchors encode the program organization supplied by the layout, including function interfaces and other fixed scaffold tokens, while the masked regions reserve space for the implementation generated by Stage II. We denote the anchor and implementation positions of a canvas c by $\mathcal{A}(c)$ and $\mathcal{I}(c)$, respectively. Tokens in $\mathcal{A}(c)$ remain fixed throughout Stage-II denoising.

During Stage-II training, the renderer receives the oracle layout $a^* = \mathcal{E}(y)$ and constructs the oracle canvas $c^* = \mathcal{C}(a^*)$. At inference, it instead receives the layout $\hat{a}$ predicted by Stage I and constructs $\hat{c} = \mathcal{C}(\hat{a})$.

Because implementation spans have variable lengths, we align each ground-truth span u_k of length $\ell_k \le b_k$ to its allocated capacity using `[NULL]` tokens:

$$\bar{u}_k = u_k \oplus \texttt{[NULL]}^{\,b_k-\ell_k}, \tag{9}$$

where $\oplus$ denotes sequence concatenation. The resulting aligned target is

$$\bar{y} = A_0 \bar{u}_1 A_1 \cdots A_{K-1} \bar{u}_K A_K. \tag{10}$$

Thus, the predicted layout determines the visible boundary conditions and available capacity of each implementation region, while Stage II remains responsible for generating the concrete code within those regions.

At inference, each implementation span is initialized with `[MASK]` tokens and completed under the fixed anchors.

3.4 Stage II: Anchor-Aware CAD Denoising

Conditioned on an anchored canvas c, Stage II models the remaining implementation distribution

$$p_\psi\left(\bar{y}_{\mathcal{I}(c)} \mid x, c\right), \tag{11}$$

where $\mathcal{I}(c)$ denotes the implementation positions of the canvas. Although the layout provides shared interfaces and region boundaries, the remaining spans are still coupled through parameters, variable bindings, intermediate solids, and downstream build logic. Stage II therefore uses anchor-aware denoising views to train both local implementation recovery and coordination across distant program regions.

During training, Stage II operates on the oracle canvas $c^* = \mathcal{C}(a^*)$ introduced above. Each denoising view $v \in \mathcal{D}(c^*)$ selects a subset of implementation positions $\mathcal{I}_v \subseteq \mathcal{I}(c^*)$ that are eligible for corruption, while all construction anchors remain visible. Given $t \sim \mathcal{U}(0,1)$, we independently mask eligible positions with probability t:

$$\mathcal{B}_{v,t} = \{i \in \mathcal{I}_v \mid \epsilon_i < t,\ \epsilon_i \sim \mathcal{U}(0,1)\}, \tag{12}$$

yielding the corrupted canvas

$$\tilde{y}^{(v,t)} = \mathrm{Mask}\left(\bar{y}; \mathcal{B}_{v,t}\right). \tag{13}$$

The Stage-II objective is

$$\mathcal{L}_{\mathrm{II}} = -\mathbb{E}_{(x,\bar{y}),v,t}\left[\sum_{i \in \mathcal{B}_{v,t}} w(t) \log p_{\theta+\Delta_\psi^{\mathrm{II}}}\left(\bar{y}_i \mid x, \tilde{y}^{(v,t)}\right)\right], \tag{14}$$

where $\Delta_\psi^{\mathrm{II}}$ is the Stage-II LoRA adapter and $w(t)$ is the diffusion reweighting term. The loss is evaluated only on masked implementation positions. Construction anchors are never corrupted, while unmasked implementation tokens remain visible as bidirectional context.

We organize the denoising views into two categories according to their dependency scope. *Single-block views* target one local program region while preserving the remaining canvas as context. They include function-body completion, symbol-consistency recovery, CadQuery API argument recovery, and syntax-sensitive local completion. *Cross-block views* jointly target semantically related regions to expose longer-range dependencies. They include global-span completion, parameter closure, joint variable-flow and reverse-dependency recovery, and runtime build-order reconstruction. In particular, global-span completion masks implementation regions jointly, exposing the model to an inference-like setting in which the implementation remains unresolved while the anchors are fixed. All views share the same Stage-II adapter and reconstruction objective.

At inference, Stage II receives the predicted canvas $\hat{c} = \mathcal{C}(\hat{a})$ from Stage I. All implementation positions are initialized with `[MASK]`, while the predicted anchors remain fixed throughout denoising. The model then iteratively completes the implementation spans conditioned on the natural-language requirement and the anchored canvas. After denoising, the generated spans are postprocessed as described above and reassembled with the fixed anchors to produce the final CadQuery program.

Additional specifications of the eight denoising views and implementation details are provided in the supplementary material.

4 CADConstruct Benchmark

We introduce **CADConstruct**, a construction-aware benchmark for natural-language-to-CadQuery generation. CADConstruct is designed to provide explicit supervision and controlled evaluation for executable parametric CAD program generation. Each example contains a dimension-grounded natural-language requirement, an executable CadQuery program, and reference geometry obtained by executing the program.

Existing natural-language-to-CadQuery benchmarks commonly represent target programs as relatively monolithic scripts, where parameter definitions, feature construction, and build logic are interleaved. CADConstruct does not introduce a new CAD language or a method-specific program format; instead, it standardizes the same executable CadQuery programs into a conventional modular Python organization. Each target contains a parameter region, feature-specific helper functions, a structured `build_model` routine, and a final result binding. This normalization makes construction boundaries explicit, reduces incidental variation in code organization, and provides a clearer representation for composing multi-step CAD operations, while leaving concrete parameters, function

Method	Easy				Medium				Hard			
	IoU↑	CD↓	Feat. F1↑	Exec.↑	IoU↑	CD↓	Feat. F1↑	Exec.↑	IoU↑	CD↓	Feat. F1↑	Exec.↑
Autoregressive Models												
InternLM2.5-7B	12.47	6.58	31.22	31.50	5.72	8.74	24.86	16.00	1.80	12.19	11.64	5.50
Llama-3.1-8B	15.24	6.32	36.74	37.75	7.81	8.41	30.94	20.25	3.17	11.77	15.46	8.25
Qwen2.5-Coder-7B	25.48	4.86	54.87	55.25	15.20	6.68	47.33	34.75	6.71	9.91	27.84	14.50
Qwen3.5-9B	23.64	4.92	49.82	52.50	12.91	5.88	41.75	31.00	6.42	9.04	24.58	13.75
CAD-RL	46.82	1.59	78.66	69.00	29.46	3.06	66.72	52.50	13.16	6.08	45.23	25.75
Qwen2.5-Coder-7B*	67.38	0.38	90.71	81.00	45.29	1.14	85.69	76.25	17.88	1.81	65.91	37.00
CAD-RL*	70.81	0.31	93.23	85.75	51.93	0.67	87.06	78.75	23.45	1.25	75.80	40.00
Diffusion Models												
Dream-Coder-7B	21.70	4.91	43.44	50.75	12.12	7.75	46.21	31.50	5.84	10.02	25.75	12.50
Dream-Coder-7B*	61.15	0.52	86.01	80.25	43.81	1.40	84.24	74.50	16.67	2.30	64.10	32.75
Ours												
CADiff*	**73.57**	**0.23**	**96.56**	**89.50**	**57.46**	**0.42**	**91.17**	**81.75**	**30.25**	**0.85**	**89.86**	**42.25**

Table 1: Performance comparison on CADConstruct. * denotes models trained on CADConstruct. IoU, Feat. F1, and Exec. are reported in percentages; CD denotes mean Chamfer Distance. Best results are shown in bold and second-best results are underlined.

bodies, variable flow, and runtime composition to be generated. The complete executable program remains the prediction target for all methods.

CADConstruct is generated through a procedural parametric modeling pipeline. For each example, the generator samples a CAD family and topology variant, together with latent component and operation structures, geometric parameters, and construction intents. These factors jointly determine the natural-language requirement and its executable CadQuery program, producing substantial variation in geometry, feature composition, operation dependencies, and program implementation. CADConstruct spans 34 part families and 85 topology variants. All generated programs are parsed, executed, and filtered through syntax, geometry, length, and duplicate checks.

CADConstruct contains 32,114 training examples and a balanced test set of 1,200 examples, with 400 examples at each of three difficulty levels: *easy*, *medium*, and *hard*. Difficulty is assigned by the data generator from properties of the target design, including component complexity, the number of construction intents, operation depth, and longer-range or multi-plane dependencies. We evaluate predictions at three complementary levels: code executability with Exec., CAD feature recovery with Feature F1, and geometric fidelity with Intersection-over-Union (IoU) and Mean Chamfer Distance (Mean CD). Details are provided in the appendix.

5 Experiments

5.1 Experimental Setup

Datasets and metrics. We evaluate on CADConstruct and additionally perform zero-shot cross-dataset evaluation on the natural-language test split of ExeCAD (Niu et al., 2026). On CADConstruct, we report executability (Exec.), coarse CAD feature F1 (Feat. F1), Intersection-over-Union (IoU), and Chamfer Distance (CD). Non-executable predictions receive Exec. $= 0$ and IoU $= 0$, while Feature F1 is computed independently from the predicted Python AST and therefore can remain nonzero for non-executable programs. Detailed metric definitions are provided in the appendix.

Implementation. CADiff uses Dream-Coder-7B (Xie et al., 2025) as the frozen backbone for both stages, with separate LoRA adapters for layout prediction and implementation denoising. Stage I is trained on natural-language–layout pairs, while Stage II is trained on oracle anchored canvases extracted from the same target programs and receives Stage-I predicted canvases at inference. Both stages are trained for three epochs. At inference, Stage I predicts the construction layout, the deterministic renderer builds the anchored canvas, and Stage II denoises the remaining implementation spans under fixed anchors. Details are provided in the supplementary material.

Table 2: Ablation on the full CADConstruct test set. S1 enables Stage-I anchoring; Single and Cross enable the two Stage-II view families. Feat. F1 measures coarse CAD feature-count agreement. IoU, Feat. F1, and Exec. are percentages.

S1	S2		IoU↑	CD↓	Feat. F1↑	Exec.↑
	Single	Cross				
×	×	×	13.22	7.56	38.47	31.58
✓	×	×	25.84	4.38	61.25	45.67
×	✓	✓	40.54	1.41	78.12	62.50
✓	×	✓	<u>51.36</u>	<u>0.67</u>	89.74	68.42
✓	✓	×	48.92	0.86	<u>90.31</u>	<u>69.25</u>
✓	✓	✓	**53.76**	**0.50**	**92.53**	**71.17**

Table 3: Zero-shot evaluation on the natural-language test split of ExeCAD. Task-specific training uses only CADConstruct, with no ExeCAD training, validation, model selection, or adaptation. CAD-RL denotes our re-implementation trained on CADConstruct rather than the original ExeCAD-trained model. IoU and Exec. are percentages; CD denotes mean Chamfer Distance.

Method	IoU↑	CD↓	Exec.↑
Qwen2.5-Coder-7B	44.80	83.20	61.25
Dream-Coder-7B	41.90	90.10	58.50
CAD-RL	50.35	73.85	66.50
CADiff	**54.10**	**64.70**	**69.75**

5.2 Main Results

Table 1 compares CADiff with autoregressive and diffusion-based baselines on CADConstruct. CADiff achieves the best performance across all twelve settings. Compared with the strongest autoregressive baseline, CAD-RL*, CADiff improves IoU by 2.76, 5.53, and 6.80 points on the easy, medium, and hard subsets, respectively, while also consistently improving Feature F1, executability, and Chamfer Distance. The gains are particularly pronounced on the hard subset, where Feature F1 increases from 75.80 to 89.86, indicating stronger recovery of the intended CAD construction as program complexity increases.

Compared with Dream-Coder-7B*, which uses the same diffusion backbone and CADConstruct training data, CADiff improves IoU by 12.42, 13.65, and 13.58 points across the three difficulty levels, together with consistent gains in Feature F1 and executability. These results support the effectiveness of resolving construction organization before implementation denoising, particularly for programs with more complex dependencies.

5.3 Ablation Study

Table 2 evaluates the contributions of Stage-I anchoring and the two Stage-II denoising view families. Stage I alone substantially improves the baseline, increasing IoU from 13.22 to 25.84 and Feature F1 from 38.47 to 61.25. Stage II provides further gains: using both single-block and cross-block views without Stage I reaches 40.54 IoU and 78.12 Feature F1. When combined with Stage I, cross-block denoising gives stronger geometric performance, while single-block denoising slightly favors Feature F1 and executability.

The full CADiff configuration achieves the best result on all four metrics, reaching 53.76 IoU, 0.50 CD, 92.53 Feature F1, and 71.17 executability. Compared with using both Stage-II view families without Stage I, adding Stage-I anchoring improves IoU by 13.22 points and Feature F1 by 14.41 points. These results show that layout conditioning and the two denoising views provide complementary benefits within the complete framework.

5.4 Stage-I Reliability and Oracle Sensitivity

Figure 3 evaluates the reliability of Stage I and the sensitivity of downstream generation to layout quality. Canvas validity remains above 98% across all difficulty levels, reaching 99.75%, 99.25%, and 98.25% on Easy, Medium, and Hard, respectively. A layout that cannot be parsed, validated, or rendered into a canvas is treated as a generation failure and is not passed to Stage II. Budget coverage is similarly high at 99.24%, 98.72%, and 98.44% on the aligned subsets, indicating that the predicted layouts are generally well-formed and provide sufficient implementation capacity.

Replacing the predicted layouts with oracle layouts yields only modest IoU gains, from 73.57 to 75.10 on Easy, 57.46 to 59.00 on Medium, and 30.25 to 32.00 on Hard. Together, these results

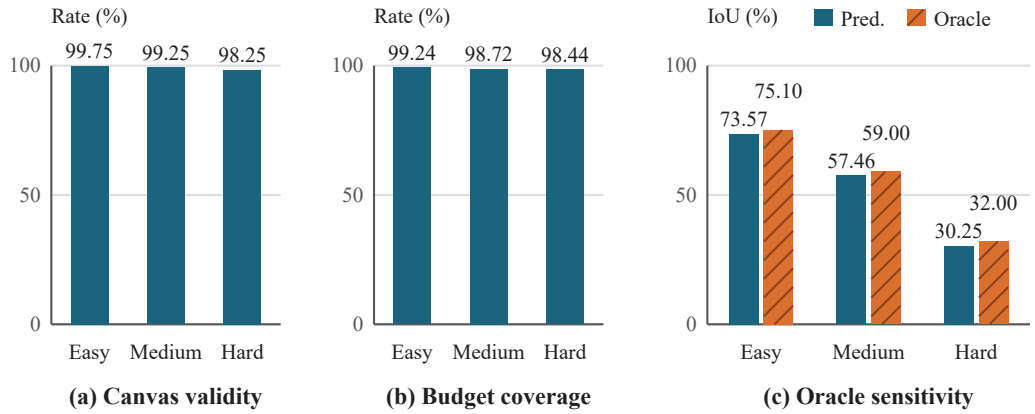

Figure 3: Stage-I reliability and downstream oracle sensitivity on CADConstruct. **(a)** Canvas validity over all 400 examples per difficulty, requiring successful layout parsing, validation, and rendering. **(b)** Program-level budget coverage on the 397, 392, and 384 examples with aligned predicted and reference spans in the Easy, Medium, and Hard splits, respectively; all reference spans must fit their allocated capacities. **(c)** Mean IoU using predicted versus reference-derived oracle layouts under the same Stage-II model and inference configuration. All values are percentages.

indicate that Stage I already provides reliable structural guidance, while further improvements in layout prediction offer limited but consistent room for downstream gains.

5.5 Cross-Dataset Evaluation

To assess cross-dataset transfer, we conduct zero-shot evaluation on the natural-language test split of ExeCAD (Niu et al., 2026). All task-specific models, including our re-implementation of CAD-RL, are trained exclusively on CADConstruct and evaluated with natural-language-only inputs. ExeCAD is not used for training, validation, model selection, or adaptation. This protocol therefore evaluates transfer from CADConstruct to the specified ExeCAD split, rather than comparing against the original ExeCAD-trained CAD-RL.

As shown in Table 3, CADiff achieves the best performance across all three metrics. Compared with CAD-RL, the strongest transferred baseline, CADiff improves IoU from 50.35 to 54.10 and executability from 66.50 to 69.75, while reducing Mean CD from 73.85 to 64.70, a relative reduction of 12.4%. CADiff also substantially outperforms Qwen2.5-Coder-7B and Dream-Coder-7B. These results provide evidence that the construction-first generation strategy transfers beyond CADConstruct under the reported zero-shot protocol.

6 Conclusion

We introduced CADiff, a construction-first diffusion framework for natural-language-to-CadQuery generation. Rather than resolving all program decisions at the same token level, CADiff separates shared construction organization from concrete implementation: Stage I predicts a compact construction layout, a deterministic renderer converts it into an anchored program canvas, and Stage II denoises the remaining implementation under fixed structural conditions. We also introduced CADConstruct, a construction-aware benchmark with dimension-grounded requirements and executable programs that explicitly expose construction structure. Experiments show consistent improvements over strong autoregressive and diffusion-based baselines, with larger gains on more complex programs. Component ablations support the complementary contributions of layout conditioning and the two denoising view families, while Stage-I reliability and oracle analyses show that predicted layouts are generally well formed and leave only a modest downstream gap to reference-derived layouts. Finally, zero-shot evaluation on ExeCAD shows that these improvements extend beyond CADConstruct under the evaluated cross-dataset protocol.

AI USE STATEMENT

We used generative AI tools during idea exploration, manuscript revision, organization of related literature, and code development. The authors reviewed and verified AI-assisted ideas, text, and code before incorporating them into this work. The authors retain full responsibility for the methodology, experimental findings, and final manuscript.

ETHICS STATEMENT

CADiff studies the generation of parametric CAD programs from natural-language specifications. CADConstruct is built through procedural modeling, and the additional evaluation uses the ExeCAD benchmark. The study does not recruit human participants or use personally identifiable or sensitive user data. We have not identified significant ethical concerns specific to the datasets, methods, or experiments reported here.

REPRODUCIBILITY STATEMENT

CADiff's two diffusion stages, construction-layout representation, and anchor-aware training and inference procedures are described in Section 3. Sections 4 and 5 report the benchmark construction and experimental settings. The appendix further documents canonical examples, span-budget allocation, the eight denoising views, dataset-generation procedures, metric definitions, and training hyperparameters.

REFERENCES

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces, 2023. URL https://arxiv.org/abs/2107.03006.

Akshay Badagabettu, Sai Sravan Yarlagadda, and Amir Barati Farimani. Query2cad: Generating cad models using natural language queries, 2024. URL https://arxiv.org/abs/2406.00144.

Yandong Guan, Xilin Wang, Ximing Xing, Jing Zhang, Dong Xu, and Qian Yu. Cad-coder: Text-to-cad generation with chain-of-thought and geometric reward. *arXiv preprint arXiv:2505.19713*, 2025.

Pradeep Kumar Jayaraman, Joseph G. Lambourne, Nishkrit Desai, Karl D. D. Willis, Aditya Sanghi, and Nigel J. W. Morris. Solidgen: An autoregressive model for direct b-rep synthesis, 2023. URL https://arxiv.org/abs/2203.13944.

Mohammad Sadil Khan, Sankalp Sinha, Talha Uddin Sheikh, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Text2cad: Generating sequential cad models from beginner-to-expert level text prompts, 2024. URL https://arxiv.org/abs/2409.17106.

Jiahao Li, Qingwang Zhang, Qiuyu Chen, Guozhan Qiu, Yunzhong Lou, and Xiangdong Zhou. Towards high-fidelity cad generation via llm-driven program generation and text-based b-rep primitive grounding, 2026. URL https://arxiv.org/abs/2603.11831.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 4328–4343. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/1be5bc25d50895ee656b8c2d9eb89d6a-Paper-Conference.pdf.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

Ke Niu, Haiyang Yu, Zhuofan Chen, Mengyang Zhao, Teng Fu, Bin Li, and Xiangyang Xue. From intent to execution: Multimodal chain-of-thought reinforcement learning for precise CAD code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 8160–8167, 2026. doi: 10.1609/aaai.v40i10.37763.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184. Curran Associates, Inc., 2024. doi: 10.52202/079017-4135. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/eb0b13cc515724ab8015bc978fdde0ad-Paper-Conference.pdf.

Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P. Adams. SketchGraphs: A large-scale dataset for modeling relational geometry in computer-aided design. In *ICML 2020 Workshop on Object-Oriented Learning*, 2020.

Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P. Adams. Vitruvion: A generative model of parametric cad sketches, 2022. URL https://arxiv.org/abs/2109.14124.

Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6772–6782, October 2021.

Yuchuan Wu, Ke Niu, Haiyang Yu, Zhuofan Chen, Xiangyang Xue, and Bin Li. Itercad: Iterative program repair for cad code generation from orthographic views, 2026. URL https://arxiv.org/abs/2608.24020.

Haoyang Xie and Feng Ju. Text-to-cadquery: A new paradigm for cad generation with scalable large model capabilities, 2025. URL https://arxiv.org/abs/2505.06507.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.

Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. Skexgen: Autoregressive generation of cad construction sequences with disentangled codebooks. In *International Conference on Machine Learning*, pp. 24698–24724. PMLR, 2022.

Xiang Xu, Joseph Lambourne, Pradeep Jayaraman, Zhengqing Wang, Karl Willis, and Yasutaka Furukawa. Brepgen: A b-rep generative diffusion model with structured latent geometry. *ACM Transactions on Graphics (TOG)*, 43(4):1–14, 2024.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025.

A ADDITIONAL DETAILS OF CADIFF

A.1 CANONICAL PROGRAM REPRESENTATION AND CONSTRUCTION LAYOUT

Program regions. We operate on the standardized CadQuery programs described in Section 4. A program contains top-level imports, a module-scope parameter region, an ordered list of helper functions, a `build_model` function, and a final result binding. The parameter region includes both numerical assignments and derived expressions, such as offsets, point lists, and dimensions computed from other parameters. Helper bodies implement geometric operations; the build routine composes these operations into the final object.

The decomposition preserves imports, function signatures, helper return bindings, and the final result binding as anchors. Assignments preceding the first top-level function form the parameter span. Each helper body, excluding its terminal return anchor, forms another implementation span. The entire `build_model` body, including its return statement, forms the final span. Nested functions and intermediate variables remain inside their enclosing implementation spans.

Layout fields and serialization. The construction layout is serialized as JSON. Table 4 summarizes the fields used to render the canvas. Each entry in `functions` records a helper name, ordered arguments, a return binding, and a body capacity. The `build_model` descriptor is stored separately. Serialization preserves the order of helper declarations and uses a consistent field order, so equivalent records do not acquire incidental variation from JSON formatting.

Layout field	Visible scaffold information	Implementation left to Stage II
`imports`	Import statements	No implementation span
`global_budget`	Position and capacity of the parameter region	Values, derived expressions, and parameter bindings
`functions`	Ordered helper signatures, return bindings, and body capacities	Geometric operations and local variable flow inside each helper
`build_model`	Build signature and body capacity	Complete build body, including calls, intermediate bindings, and return
`final`	Final top-level result binding	No implementation span

Table 4: Construction-layout fields and the corresponding division of responsibility between the anchored scaffold and implementation spans. Schema metadata and record identifiers are omitted.

The schema and rendering rules are shared across examples, while Stage I predicts the instance-specific interfaces and capacities. The layout does not contain a CAD-family label that specifies the implementation. In particular, the order of entries in `functions` determines *declaration order*; the sequence of executed calls remains part of the `build_model` implementation generated by Stage II.

Deterministic extraction. The extractor parses a reference program with Python's `ast` module and uses source order and AST ranges to locate region boundaries. A helper's terminal return is retained as an anchor, whereas the build routine's return stays inside its implementation span.

A.2 ANCHORED CANVAS CONSTRUCTION AND SPAN BUDGETING

Rendering. For H helpers and one parameter region, the renderer creates $K = H + 2$ implementation spans: the parameter span, the H helper bodies in declaration order, and the build body. Each span receives b_k mask positions, interleaved with the anchor sequences as in Equation 8. Signatures and helper return statements remain outside these positions. The canvas is an intermediate token representation; Python syntax is checked after completion and assembly.

Reference capacities. For training, span length ℓ_k is measured with the Dream-Coder tokenizer. The preparation procedure adds bounded slack and rounds the initial capacity to a multiple of eight:

$$\delta_k = \min(64, \max(4, \lceil 0.1\ell_k \rceil)), \qquad b_k^{\text{init}} = 8 \left\lceil \frac{\ell_k + \delta_k}{8} \right\rceil . \tag{15}$$

View	Eligible implementation content	Context and reconstruction focus
	Single-block views	
Function-body completion	One helper-function body	Fixed interface and other spans support recovery of a local feature implementation.
Symbol-consistency recovery	Parameter or variable occurrences in the selected local region	Visible expressions and bindings constrain consistent symbol use.
CadQuery API argument recovery	Argument expressions of selected CadQuery calls	API names and surrounding code constrain dimensions, planes, and transformations.
Syntax-sensitive local completion	Structurally complex expressions in a local region	Surrounding statements support reconstruction of calls, comprehensions, and geometry expressions.
	Cross-block views	
Global-span completion	All implementation spans	Only anchors are guaranteed to remain visible; unresolved regions are reconstructed jointly.
Parameter closure	The parameter region	Helper and build implementations expose downstream uses of parameters.
Joint variable-flow and reverse-dependency recovery	Related producer regions	Later consumers and build logic constrain upstream values and intermediate-solid flow.
Runtime build-order reconstruction	The complete `build_model` body	Visible helper implementations and interfaces constrain calls, bindings, and composition order.

Table 5: The eight Stage-II denoising views. Entries describe eligible regions; actual corruption within them is sampled at noise level t. Every view preserves construction anchors and uses the same Stage-II adapter and reconstruction objective.

The slack accommodates variation in implementation length. If the combined prompt, anchors, and allocated spans exceed the Stage-II length limit, capacity adjustment removes only surplus padding, maintaining $b_k \geq \ell_k$.

This length-based calculation prepares reference training layouts. At inference, the capacities come from the layout predicted by Stage I; the reference implementation and its length are unavailable. Stage II generates within these predicted capacities.

Alignment and postprocessing. Each reference span is padded to its allocated capacity using the single-token placeholder denoted by `[NULL]`, yielding the aligned target in Equation 10. As an arithmetic example, $\ell_k = 81$ gives $\delta_k = 9$ and $b_k^{\text{init}} = 96$; without capacity adjustment, this span receives 15 trailing `[NULL]` tokens. Padding is part of the reconstruction target and can be corrupted wherever its span is eligible for denoising.

A.3 Anchor-Aware Denoising Views

Eligible regions and stochastic corruption. A view specifies the set $\mathcal{I}_v$ of implementation positions eligible for corruption. Given this set, each position is masked independently with probability t, as in Equation 12. Selecting a span therefore does not imply masking every token in that span at every training step. Anchors are always visible, and implementation positions outside $\mathcal{I}_v$ provide uncorrupted context. Supervision applies only to the positions actually masked in $\mathcal{B}_{v,t}$.

Table 5 describes the eight views using the names and dependency scopes introduced in the main paper. Single-block views focus on a local implementation region. Cross-block views emphasize the relationship between a region and the wider program, including cases where one producer or build span is reconstructed from context in other spans.

Shared model and inference connection. The views are training conditions for one Stage-II model, rather than separate adapters or a sequence of eight inference passes. A view changes which implementation positions can be corrupted; it does not alter the reference program or its layout. In the global-span view, every implementation position is eligible, so training includes joint recovery across the whole canvas. At inference, all implementation positions start masked and are completed together under the predicted anchors.

A.4 Training and Inference Procedures

Training. The two stages use the same frozen Dream-Coder-7B backbone with separate LoRA adapters. Stage I learns the serialized reference layout conditioned on the natural-language requirement. Stage II learns implementation reconstruction from reference layouts and aligned reference programs. Both stages sample $t \sim U(0, 1)$ and use the time weighting $w(t) = 1/t$; their supervised token sets are those specified in Equations 7 and 14. The optimizer and adapter settings are summarized in Table 11. Both stages are trained for three epochs.

The training procedure can be expressed as the following two branches for a natural-language–program pair (x, y):

1. Extract reference layout a^* and implementation spans u^* from y; serialize the layout as $z_0 = \mathcal{S}(a^*)$.
2. **Stage I:** sample t, corrupt tokens in z_0, and update the Stage-I adapter using the masked layout reconstruction loss.
3. **Stage II:** render the reference canvas and align the spans with `[NULL]` padding to form $\bar{y}$.
4. Select a training view v, obtain $\mathcal{I}_v$, sample t, and corrupt only eligible implementation positions.
5. Update the Stage-II adapter using the loss on $\mathcal{B}_{v,t}$; anchors and all unmasked tokens remain context.

The branches describe separate adapter training. They do not require passing sampled Stage-I predictions into Stage-II training.

Inference. The input is the natural-language requirement x. Neither reference code nor an oracle layout is provided. The inference procedure is:

1. Initialize a masked layout sequence and denoise it with the Stage-I adapter conditioned on x.
2. Deserialize and validate the layout $\hat{a}$. If layout parsing, validation, or canvas rendering fails, return a generation failure, as specified in Section 5.4.
3. Render $\hat{c} = \mathcal{C}(\hat{a})$, initializing every implementation position with `[MASK]` and retaining the predicted anchors.
4. Denoise the implementation positions with the Stage-II adapter, keeping anchors fixed throughout generation.
5. Postprocess the completed spans, reassemble the program, and apply the syntax and execution checks in Section C.

Stage-II commitment schedule. The sampler uses a linear time grid from 1 to 10^{-3}. At a step from t_i to $s_i < t_i$, let m_i be the number of currently masked implementation positions. Under entropy-based selection, the sampler predicts token distributions at these positions and commits the lowest-entropy positions, with commitment count

$$n_i = \left\lfloor m_i \left(1 - \frac{s_i}{t_i}\right) \right\rfloor. \tag{16}$$

All remaining masked positions are committed at the final step.

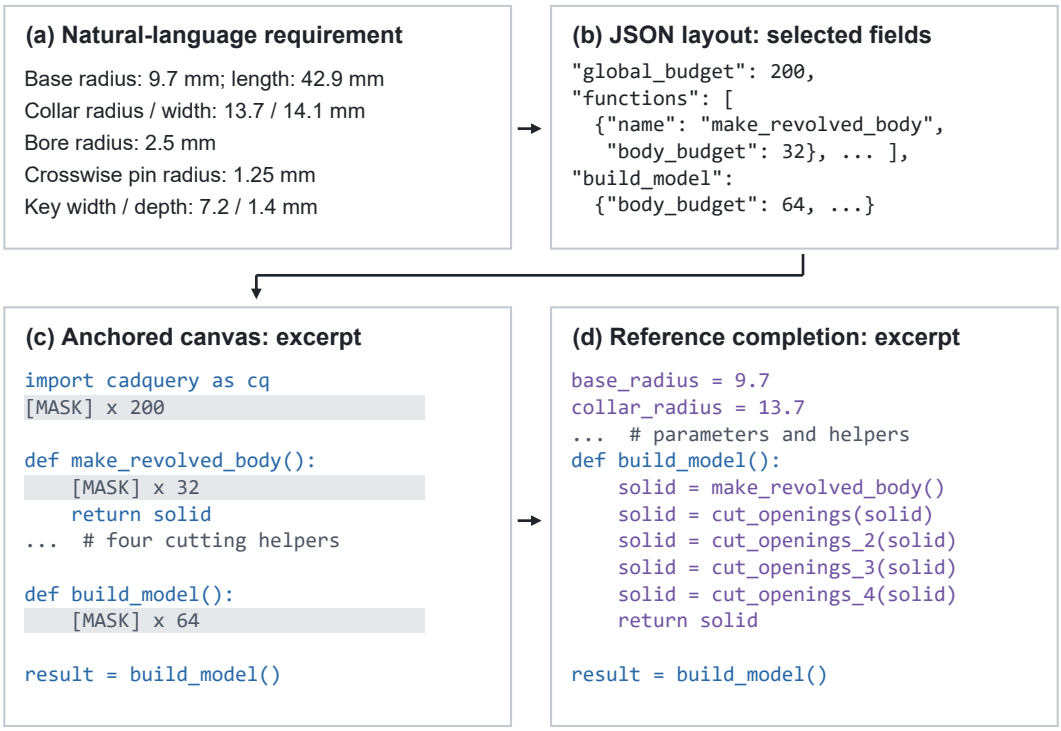

Figure 4: A reference example of the CADiff representation. **(a)** Selected dimensions from the natural-language requirement. **(b)** Selected JSON layout fields. **(c)** The rendered canvas preserves signatures and result bindings while reserving implementation capacity. **(d)** Reference parameter assignments and the complete build body fill the corresponding regions. Blue denotes anchors, gray denotes mask positions, and purple denotes implementation content. Ellipses indicate omitted fields or code regions; capacities are taken from the stored record. All displayed completions come from the reference program.

A.5 End-to-End Worked Example

We use the reference example `cadq_easy_stepped_shaft_001394` to illustrate the mapping between a requirement, a stored layout, and a complete program. Figure 4 shows selected fields and code regions from the same record. This is a reference decomposition and reassembly example; its displayed completion is reference code.

Natural-language requirement.

> Create a stepped shaft in millimeters. Start with the base radius set to 9.7 mm as the reference body for the remaining features. Add the collar 13.7 mm in radius and 14.1 mm wide centered along the shaft length. Set the shaft length to 42.9 mm as the revolved axial body. Drill the bore with a 2.5 mm radius through the rotation axis. Set the locating pin radius to 1.25 mm crosswise through the shaft. Cut the key 7.2 mm wide and 1.4 mm deep across the collar face.

Layout and canvas. The record contains five helpers: `make_revolved_body`, `cut_openings`, and `cut_openings_2` through `cut_openings_4`. The first helper takes no arguments; the remaining helpers each consume and return `solid`. The parameter, helper-body, and build-body capacities, in canvas order, are

$$\mathbf{b} = (200, 32, 64, 72, 72, 72, 64).$$

There are seven implementation spans and 576 allocated implementation positions. Import statements, helper signatures and terminal returns, the build signature, and `result = build_model()` lie outside these spans.

Implementation and reassembly. The parameter span supplies dimensions and derived quantities, including `half_length`, `half_collar`, and the revolved profile points. The helpers create the revolved body and perform the bore, crosswise cuts, and key operation. The build span contains the helper calls, successive `solid` assignments, and its own return statement. These calls are implementation tokens, even though the helper declarations are anchors. Reassembling the stored reference spans with their anchors reproduces the reference program's Python AST.

The same example also illustrates the training views. Function-body completion can select the body of `make_revolved_body` while retaining its interface and the rest of the program. Parameter closure can select the parameter region while preserving the geometric calls that consume those parameters. Runtime build-order reconstruction can select the entire build body while leaving the helper implementations visible. In each case, the selected positions are then independently corrupted according to the sampled noise level; the canvas boundaries remain fixed.

B CADConstruct Benchmark Details

B.1 Dataset Records and Joint Generation

Each CADConstruct record pairs a dimension-grounded natural-language requirement with a complete CadQuery program and reference geometry obtained by executing that program. Family and topology identifiers describe the generating design, while the requirement supplies the input to the generation task. The prediction target is the complete executable program, including its numerical parameters, feature implementations, and build routine.

The procedural generator samples a family, a topology variant, geometric parameters, and construction intents. A latent component graph describes the participating components and their semantic relationships. An operation DAG describes the operations that construct and combine them. The requirement and the program are emitted from this shared state, so their dimensions and construction instructions refer to the same design. The reference geometry is produced by program execution rather than generated independently from the text.

Component and operation structure. Component nodes contain an identifier and semantic type. Edges identify their source and target components and a relation, such as reinforcement or a right-angle connection. Operation records contain an operation identifier, its type, and the relevant input and output components. These structures organize dataset generation. At inference, CADiff receives the natural-language requirement and predicts its own construction layout; the latent component graph and operation DAG are not supplied as additional inputs.

Program organization. Generated programs follow the modular representation in Section A.1. Parameters and derived expressions appear at module scope. Helpers implement local features, and `build_model` binds intermediate solids and composes the operations. Repeated features can be realized through point lists, counts, and pattern operations. The generator uses sketches and primitives including boxes, circles, rectangles, slots, polygons, polylines, splines, and arcs, together with operations such as extrusion, revolution, lofting, translation, rotation, union, cut, chamfer, and fillet. Variation in the sampled state changes dimensions, feature multiplicity, helper composition, and the concrete operation sequence.

B.2 Part Families, Topology Variants, and Visual Examples

CADConstruct contains 34 part families and 85 observed topology variants. Table 6 lists the family taxonomy, its difficulty assignment, and the number of observed variants. These counts describe structural categories, rather than the frequency of examples in each category. A family identifies a broad part type; a topology variant distinguishes alternative feature or component arrangements within that family. Parameter sampling introduces further geometric variation within a variant.

Figure 5 shows eight reference designs at each difficulty level. The examples illustrate geometric and compositional variation, including mounting parts, shafts, flanges, enclosures, supports, and multi-component fixtures. The renderings complement the taxonomy; they are examples from the benchmark rather than predictions used to compare model performance.

Family	Difficulty	Variants
clevis_fork_link	Easy	2
dovetail_key_rail	Easy	4
hex_spacer	Easy	2
mounting_plate	Easy	2
polygon_cap	Easy	5
round_flange	Easy	2
spacer_bar	Easy	2
stepped_shaft	Easy	1
u_channel_bracket	Easy	4
wedge_ramp_stop	Easy	1
angle_gusset_bracket	Medium	2
arc_slot_flange	Medium	3
cross_tee_plate	Medium	2
dovetail_slide_block	Medium	4
electronics_cover	Medium	2
h_frame_plate	Medium	4
open_service_tray	Medium	4
oval_access_cover	Medium	2
polyline_cam_plate	Medium	2
pulley_wheel	Medium	3
radial_index_arm	Medium	6
ring_clamp	Medium	1
sensor_bracket	Medium	2
spline_index_plate	Medium	2
tapered_adapter	Medium	2
arch_bridge_fixture	Hard	1
boxed_service_tray	Hard	2
projection_fixture	Hard	3
pump_housing	Hard	3
revolved_coupler	Hard	3
service_manifold	Hard	1
slide_carriage	Hard	1
truss_frame_fixture	Hard	4
wedge_indexing_jig	Hard	1
Total: 34 families		85

Table 6: CADConstruct family and topology inventory. Easy, Medium, and Hard contain 10, 15, and 9 families, respectively, spanning 25, 41, and 19 observed topology variants. Variant counts are not sample counts.

B.3 Validation, Filtering, and Exact Deduplication

Candidate records undergo syntax, execution, geometry, length, and duplicate checks. Python parsing verifies that the emitted target is a syntactically valid program. Execution checks whether it produces a usable CAD object, and reference geometry is exported for subsequent evaluation. Length filtering ensures that retained records can be prepared for the model's supported sequence lengths.

Exact duplicate checks consider target-code hashes, input-text hashes, and parameter signatures. The final audit reports 33,314 distinct values for each of these signatures, with zero exact overlap between the training and test partitions under the same checks. These tests identify repeated records under their respective definitions. They do not establish disjointness of de-parameterized code structures, part families, or topology variants.

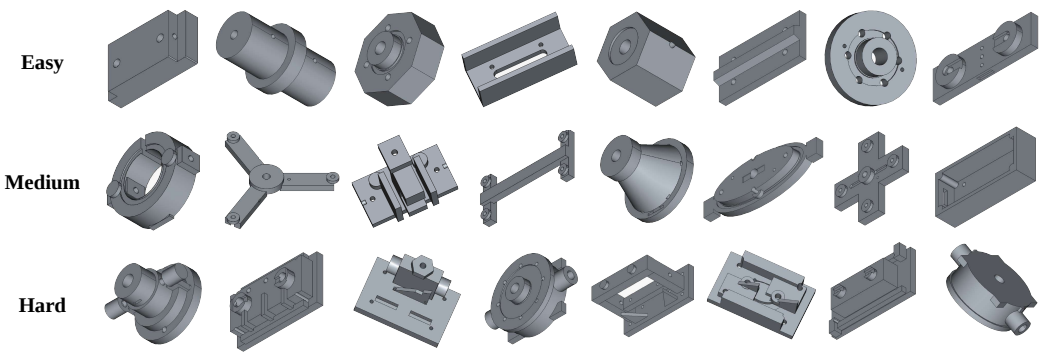

Figure 5: Representative reference geometries from CADConstruct, grouped into Easy, Medium, and Hard examples from top to bottom. Each row shows eight designs illustrating variation in part geometry, feature composition, and construction complexity. Display scales are adjusted for visibility and do not encode physical size.

B.4 Difficulty and Dataset Partitions

The final dataset contains 32,114 training records and 1,200 test records, for a total of 33,314. The test set is balanced across three difficulty levels, with 400 records per level (Table 7). Difficulty is determined by the generator from the target design, before any model is evaluated. Its criteria concern component complexity, construction intents, operation depth, and longer-range or multi-plane dependencies.

Partition	Records
Training	32,114
Test: Easy	400
Test: Medium	400
Test: Hard	400
Test total	1,200
Dataset total	33,314

Table 7: Final CADConstruct partition sizes. The three test subsets are disjoint and equally sized.

Easy designs involve relatively simple component composition and a small number of construction intents. Medium designs combine more features and dependencies. Hard designs require more extensive composition, operation sequencing, or multi-plane reasoning. The evaluation therefore groups examples by properties of the reference design, rather than by observed model success or a learned difficulty score. The balanced test split supports separate comparisons at each level; it does not imply that training records or part families are equally distributed across levels.

C Evaluation Protocol

C.1 Evaluation Units, Executability, and Failures

On CADConstruct, a test example consists of a natural-language requirement and its reference program and geometry. A generated program is assessed for executability, coarse feature-count agreement, and geometric fidelity. IoU, Feature F1, and Exec. are displayed as percentages; Chamfer Distance is displayed separately. Each difficulty subset contains 400 evaluation requests. Failed requests remain part of the denominator for executability and IoU. The evaluator rejects the explicit invalid pattern `.circle(0)`. The execution wrapper uses a 20-second timeout per sample. Let $e_i \in \{0, 1\}$ indicate this condition; the reported executability is

$$\text{Exec.} = \frac{100}{N} \sum_{i=1}^{N} e_i. \tag{17}$$

Layout parsing, schema validation, or canvas-rendering failure in Stage I terminates that generation request before Stage II, as described in Section 5.4. Syntax errors, runtime exceptions, timeouts, missing results, and empty results also receive Exec. $= 0$ and IoU $= 0$.

Outcome	Exec.	IoU
Stage-I layout or canvas failure	0	0
Python syntax/runtime error, timeout, or missing/empty result	0	0
Executable result with failed or empty boolean comparison	1	0
Executable result with successful geometric comparison	1	Computed

Table 8: Per-example execution and IoU outcomes before conversion to percentages. Feature F1 is computed independently from program syntax; CD additionally requires usable surface samples.

C.2 Coarse CAD Feature F1

Feature F1 compares feature counts extracted statically from the predicted and reference Python ASTs. Identifiers, attributes, and call names are mapped to seven coarse groups: holes/bores, slots/cutouts, pads/bosses/locators, supports/ribs, ports/seals, edge details, and patterns/repetition. Let $c_p(g)$ and $c_y(g)$ denote the counts for group g. Define

$$O = \sum_g \min(c_p(g), c_y(g)), \qquad n_p = \sum_g c_p(g), \qquad n_y = \sum_g c_y(g). \tag{18}$$

For positive denominators, precision and recall are O/n_p and O/n_y. The corresponding per-example F1 can be written without division by either individual count:

$$F_1 = \begin{cases} 1, & n_p + n_y = 0, \\ \dfrac{2O}{n_p + n_y}, & n_p + n_y > 0. \end{cases} \tag{19}$$

Thus, one empty counter and one nonempty counter give zero overlap, whereas the empty–empty case receives one. An unparsable prediction produces an empty predicted counter under the stated AST protocol.

This metric measures coarse feature-count agreement. It does not verify dimensions, feature placement, execution order, or the correctness of intermediate-solid flow. A parseable program that fails during execution can consequently obtain nonzero Feature F1. These properties explain why Feature F1 is reported alongside executability and geometric metrics.

C.3 Intersection-over-Union

For executable predicted and reference B-rep shapes A and B, the geometric evaluator uses OpenCascade boolean intersection and union:

$$\mathrm{IoU}(A, B) = \frac{\mathrm{Vol}(A \cap B)}{\mathrm{Vol}(A \cup B)}. \tag{20}$$

The resulting value is clipped to $[0, 1]$. The comparison uses the program-generated geometry, without learned alignment or voxelization. An unsuccessful boolean comparison, an empty prediction, or a non-executable prediction receives zero. The reported split-level IoU is $100N^{-1}\sum_i \mathrm{IoU}_i$, including these failures.

C.4 Chamfer Distance and Surface Sampling

On CADConstruct, executable Workplanes are exported to STL. Surface point sets are sampled with `trimesh`, using 1,000 points per shape. Meshes with no vertices or more than 100,000 faces are rejected. For sampled point sets P and Q, let $d(p, Q) = \min_{q \in Q} \|p - q\|_2$. The raw symmetric squared Chamfer Distance is

$$\mathrm{CD}_{\mathrm{raw}}(P, Q) = \frac{1}{|P|}\sum_{p \in P} d(p, Q)^2 + \frac{1}{|Q|}\sum_{q \in Q} d(q, P)^2. \tag{21}$$

This definition sums the two directed terms. Coordinate normalization and any reporting multiplier affect the numerical scale of CD, whereas execution, export, and sampling failures determine which distances are available for aggregation. A missing geometry is not a zero-distance match.

C.5 Stage-I Diagnostics and Oracle Evaluation

Canvas validity. Let V_i indicate that a Stage-I output can be deserialized, satisfies the layout schema, and is successfully rendered into an anchored canvas. For difficulty d, the validity rate is

$$\text{Validity}_d = \frac{100}{N_d} \sum_{i=1}^{N_d} V_i, \qquad N_d = 400. \tag{22}$$

This checks the structural input to Stage II. It does not require the still-masked canvas to be executable Python or measure whether its eventual geometry matches the requirement.

Program-level budget coverage. Let $\mathcal{S}_d$ contain examples whose predicted and reference layouts admit a one-to-one correspondence between implementation spans. For an aligned example i, let ℓ^*_{ik} be the reference span length measured with the Stage-II tokenizer and $\hat{b}_{ik}$ the capacity supplied by the predicted canvas. Define

$$C_i = \mathbf{1}\left[\ell^*_{ik} \le \hat{b}_{ik} \text{ for every aligned span } k\right], \qquad \text{Coverage}_d = \frac{100}{|\mathcal{S}_d|} \sum_{i \in \mathcal{S}_d} C_i. \tag{23}$$

Coverage requires every reference span in a program to fit; it is not an average of token-level or span-level coverage. Its denominator is the aligned subset, which differs from the full-test denominator used for canvas validity. It measures capacity for the particular reference implementation; an alternative implementation may have a different length.

Difficulty	Test	Valid canvases	Aligned	Covered
Easy	400	399	397	394
Medium	400	397	392	387
Hard	400	393	384	378

Table 9: Counts underlying Figure 3(a,b). Validity divides valid canvases by the full test count. Budget coverage divides covered programs by the aligned count, giving 99.24%, 98.72%, and 98.44% on Easy, Medium, and Hard, respectively.

Oracle sensitivity. The oracle setting replaces the predicted construction layout with one extracted from the reference program, including its interfaces and span capacities. The Stage-II model and inference configuration remain fixed. Figure 3(c) reports predicted/oracle IoU of 73.57/75.10, 57.46/59.00, and 30.25/32.00 on Easy, Medium, and Hard. The corresponding gains are 1.53, 1.54, and 1.75 percentage points. This comparison measures downstream sensitivity to replacing the full layout.

C.6 Ablation Configurations

The ablation in Table 2 varies Stage-I anchoring and the availability of the two Stage-II view families. The flags identify configuration components rather than independently measured stages. Every row is evaluated through its final generated program, using the same metric definitions. Table 10 records the six combinations reported in the main paper.

Comparisons between rows 1 and 2, or rows 3 and 6, assess the change associated with enabling anchoring under the corresponding view configuration. Rows 4 and 5 compare the two view families in the presence of anchoring; row 6 combines them. These comparisons concern end-to-end generation quality and do not by themselves establish equal training compute or measure dependency correctness directly.

Row	S1	Single	Cross	Enabled components
1	×	×	×	Reference configuration without the three components
2	✓	×	×	Stage-I anchoring
3	×	✓	✓	Both view families, without Stage-I anchoring
4	✓	×	✓	Anchoring and cross-block views
5	✓	✓	×	Anchoring and single-block views
6	✓	✓	✓	Full CADiff

Table 10: Configuration flags for the six rows of the main ablation table. Single and Cross refer to the view families in Table 5.

C.7 Zero-Shot Evaluation on ExeCAD

The transfer experiment uses the natural-language test split of ExeCAD. All task-specific models in Table 3, including the CAD-RL re-implementation, are trained on CADConstruct. ExeCAD is not used for training, validation, model selection, or adaptation. Inputs are natural-language requirements without image conditioning. The CAD-RL entry is the CADConstruct-trained re-implementation, rather than the original ExeCAD-trained model.

This experiment evaluates transfer between the two benchmark sources. It is distinct from withholding selected CADConstruct families or topology variants during training. CD values from the two benchmarks should be compared within their respective protocols unless their normalization and aggregation conventions have been verified to match.

D Full Reproducibility Configuration

D.1 CADiff Training Configuration

Both stages use a frozen Dream-Coder-7B backbone with separate LoRA adapters. Stage I is trained on natural-language–layout pairs derived from CADConstruct, while Stage II is trained on aligned reference programs and their oracle canvases. Table 11 collects the training settings. The distinct sequence limits reflect the different targets: compact layout JSON and the longer aligned program, respectively.

Parameter	Stage I	Stage II
Backbone	Dream-Coder-7B	Dream-Coder-7B
Trainable module	LoRA-I	LoRA-II
Training epochs	3	3
Learning rate	5×10^{-5}	5×10^{-5}
Effective batch size	16	16
LoRA rank / alpha	64 / 128	64 / 128
LoRA dropout	0.01	0.01
Maximum sequence length	1024	2048
Optimizer	AdamW	AdamW
Betas	$(0.9, 0.95)$	$(0.9, 0.95)$
Weight decay	0.01	0.01
Warmup ratio	0.1	0.1
Precision	bfloat16	bfloat16
Gradient clipping	1.0	1.0
Noise sampling	$t \sim U(0, 1)$	$t \sim U(0, 1)$
Time weighting	$1/t$	$1/t$

Table 11: Training configuration for the two CADiff adapters. Maximum lengths refer to the prepared training sequences.

The Stage-II views share one adapter and one reconstruction objective. The training targets use reference layouts; inference uses layouts predicted by the Stage-I adapter. Accordingly, the Stage-I and Stage-II checkpoints must be identified separately even though their frozen backbone is shared.